Persistent

Core Java:
Stream (java.util.stream)

Persistent Interactive | Persistent University

Objectives :

At the end of this module, you will be able to:

- Understand and Construct streams
- Describing the Stream Interface
- Understand Intermediate operations of stream
- Understand Terminal operations of stream
- Filtering a collection using lambda expressions.
- Calling an existing method using a method reference
- Chaining multiple methods together
- Defining pipelines in terms of lambdas and collections
- Understand Optional class
- New methods of Stream API

What are Streams?

- A stream is “a sequence of elements from a source that supports data processing operations.”
- Streams introduced in java 8 are like **Monads**
- Streams lets you process data in a declarative way.
- Streams let you process collections with database-like operations
- Streams can leverage multi-core architectures without you having to write a single line of multithread code.

In functional programming, a monad is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together.

How Streams are different from Collections

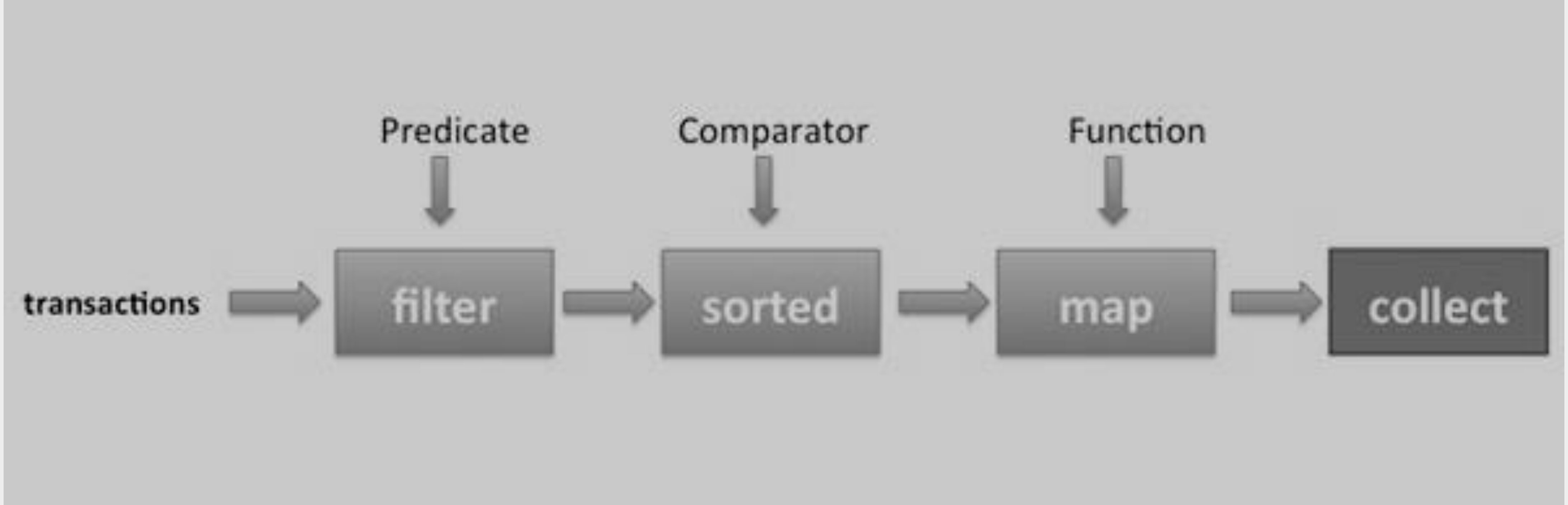
- Streams differ from collections in several ways:
  - No storage
    - A stream is not a data structure
  - Functional in nature
    - An operation on a stream produces a result, but does not modify its source.
  - Laziness-seeking
    - Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization.
    - Processing streams lazily allows for significant efficiencies;
  - Possibly unbounded
    - Collections have a finite size, streams need not.
  - Consumable
    - The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Stream Operations

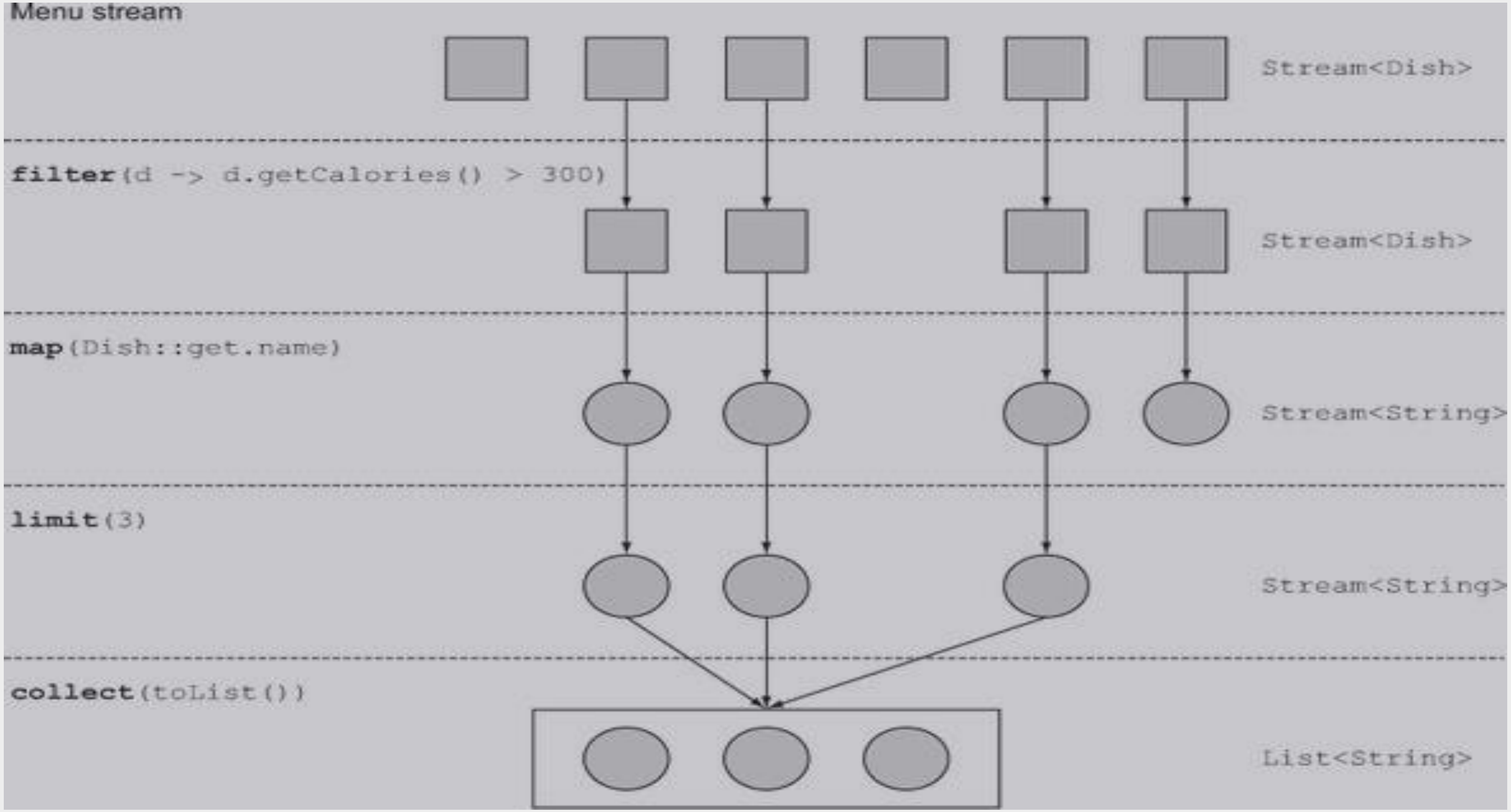
- Stream operations are divided into
  - intermediate operations
  - terminal operations
- Operations are combined to form stream pipelines
- A stream pipeline consists of a source
  - such as a Collection, an array, a generator function, or an I/O channel.

Pipeline –a sequence of aggregate operations.

First, obtain a stream from the list of transactions (the data) using the stream() method available on List. Next, several operations (filter, sorted, map, collect) are chained together to form a pipeline, which can be seen as forming a query on the data.



Processing of stream operations



Intermediate operations

- Operations like stream.filter or stream.map are intermediate operations
- They always return a new stream.
- They are always *lazy*
  - Executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.
  - Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

Intermediate operations continue….

- Intermediate operations are further divided into stateless and stateful operations.
- Stateless operations, such *as filter and map*, retain no state from previously seen element when processing a new element -- each element can be processed independently of operations on other elements.
- Stateful operations, such *as distinct and sorted*, may incorporate state from previously seen elements when processing new elements.
  - Stateful operations may need to process the entire input before producing a result.

Terminal operations

- Operations like Stream.forEach or IntStream.sum are terminal operations
- These operations may traverse the stream to produce a result or a side-effect..
- After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used
- if you need to traverse the same data source again, you must return to the data source to get a new stream.
- In almost all cases, terminal operations are *eager*, completing their traversal of the data source and processing of the pipeline before returning
- Only the terminal operations iterator() and spliterator() are not eager.

Using streams

```
public class Person {

private String personName;
private int points;
private String city;
private String gender;

public Person() {
// TODO Auto-generated constructor stub
}
… parametrized constructors and getter , setter methods

}// end of person
```

Using streams (continued…….)

```
// method which returns list of persons
List<Person> getPersons(){

Person p1=new Person("Ram",100,"pune");
Person p2=new Person("Tom",140,"pune");
Person p3=new Person("Geet",500,"mumbai");
Person p4=new Person("Preet",700,"mumbai");

List<Person> personsList=new ArrayList<Person>();
personsList.add(p1);
personsList.add(p2);
personsList.add(p4);
personsList.add(p3);

return personsList;
}
```

Using streams (continued…….)

```
// using stream operations
// Obtain stream

getPersons().stream()

// filter stream based of predicate
.filter(person->person.getPoints()>500)

// printing the filtered result
forEach(Person::printPerson);
```

Tired of Null Pointer Exceptions? Let’s Use Optional!

The null reference is the source of many problems because it is often used to denote the absence of a value. Java SE 8 introduces a new class called java.util.Optional that can alleviate some of these problems.

Characteristics of Optional class:-

- Represents a container object which may or may not contain a non-null value.
- Provides isPresent() method which will return true If a value is present
- Provides get() method to return the value.
- Additional methods that depend on the presence or absence of a contained value are provided, such as orElse() (return a default value if value not present) and ifPresent()
- *This is a* value-based *class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of Optional may have unpredictable results and should be avoided.*

Advantages of Optional class

- Null checks are not required.
- No more NullPointerException at run-time.
- We can develop clean and neat APIs.
- No more Boiler plate code

Using Optional class

```
// Creating an empty Optional:
Optional<Soundcard> sc = Optional.empty();

// Creating an Optional with a non-null value:
SoundCard soundcard = new Soundcard();
Optional<Soundcard> sc = Optional.of(soundcard);

// Using ofNullable to create an Optional object that may hold a null value:
Optional<Soundcard>sc=
Optional.ofNullable(soundcard);


// use of ifPresent
Optional<Soundcard> soundcard = ...;
 soundcard.ifPresent(System.out::println);
```

Optional class new methods

- stream()
  - If a value is present, it returns a sequential Stream containing only that value, otherwise returns an empty Stream.
- ifPresentOrElse()
  - If a value is present, performs the given action with the value, otherwise performs the given empty-based action.
- or()
  - If a value is present, returns an Optional describing the value, otherwise returns an Optional produced by the supplying function.

New methods in Stream API

- default Stream<T> takeWhile(Predicate<? super T> predicate)

Example

```
import java.util.stream.Stream;

public class Tester {

 public static void main(String[] args) {

                Stream.of("a","b","c","","e","f").takeWhile(s->!s.isEmpty()).forEach(System.out::print);

} }
```

New methods in Stream API

- default Stream<T> dropWhile(Predicate<? super T> predicate)

```
import java.util.stream.Stream;

public class Tester {
   public static void main(String[] args) {

      Stream.of("a","b","c","","e","f").dropWhile(s->
!s.isEmpty()).forEach(System.out::print);

Stream.of("a","b","c","","e","","f").dropWhile(s->
!s.isEmpty()).forEach(System.out::print);

 }
}
```

New methods in Stream API

- static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)

Example

```
import java.util.stream.IntStream;

public class Tester {

  public static void main(String[] args) {

     IntStream.iterate(3, x -> x < 10, x -> x+3).forEach(System.out::println);

   }
}
```

New methods in Stream API

- static <T> Stream<T> ofNullable(T t)

Example

```
import java.util.stream.Stream;

public class Tester {

  public static void main(String[] args) {

     long count = Stream.ofNullable(100).count();
      System.out.println(count);

     count = Stream.ofNullable(null).count();
      System.out.println(count);

  }
}
```

Summary

With this we have come to an end of our session, where we discussed about

- Stream interface
- Optional Class and its need
- Advantages of using Stream API
- Methods of Stream API

Appendix

References

Key Contacts

Reference Material : Websites & Blogs

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html

Reference Material: Books

Java SE 8 for the Really Impatient

By: Cay S Horstmann

Java 8 Lambaddas

By: Richard Warburton

Thank you!

Persistent Interactive | Persistent University